

UNIVERSIDAD NACIONAL DE INGENIERÍA

Introducción a la Computación

Internet y una mirada a Python

De la dirección en el navegador a la página · El mismo programa en C y en Python

Gerson Garrido

Semana 14

Hoy

1. Internet: paquetes, routers, direcciones IP, TCP y puertos
2. DNS: de nombres a números
3. HTTP: petición, respuesta y códigos de estado, con un servidor propio
4. HTTPS: qué protege y qué no, visto con un espía
5. Compilado frente a interpretado. ¿Por qué hay más de un lenguaje?
6. El mismo programa en pseudocódigo, en C y en Python
7. Velocidad y errores: qué se gana y qué se paga

> **Lectura**
>
> Apuntes de la semana 14. Código en `code/week-14`. El laboratorio corre en `127.0.0.1`: no necesita internet. Python es una primera mirada: no se evalúa programarlo.

SECCIÓN 01

Internet

01

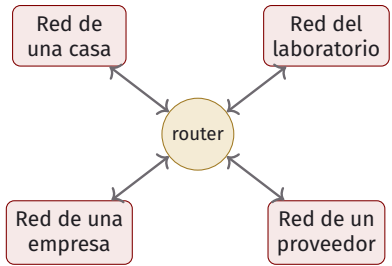


Internet es una red de redes

- Una **red**: computadoras conectadas que se mandan datos.
- **Internet**: muchas redes conectadas entre sí. Nadie la maneja entera.
- **Protocolo**: las reglas que todas siguen.

Internet no es la web

Internet es la red. La web es un servicio encima de ella, como el correo.



Paquetes y direcciones IP

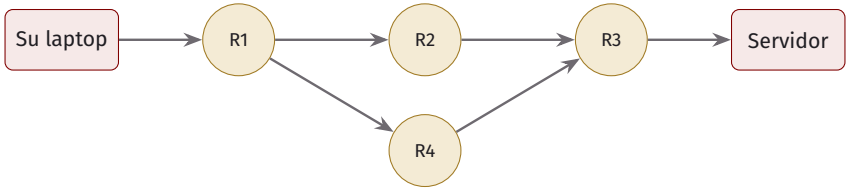
- Los datos se cortan en **paquetes** (*(packets)*). Cada uno lleva destino y origen.
- Como un libro mandado en muchos sobres. Muchas conversaciones comparten un mismo cable.

Versión	Tamaño	Ejemplo
IPv4	32 bits, cuatro números de 0 a 255	`192.168.1.20`
IPv6	128 bits, en hexadecimal	`::1`

Una dirección especial

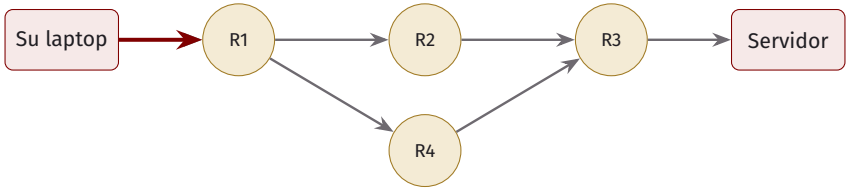
`127.0.0.1` es esta misma computadora. Esos paquetes **no salen a la red**. Por eso el laboratorio funciona sin internet.

Routers: un salto a la vez



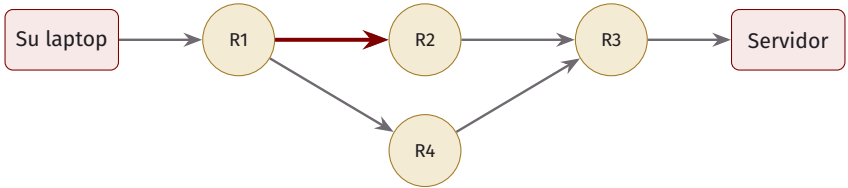
- El router mira el destino, consulta su tabla y entrega al **siguiente salto**.
- Cada router solo decide el siguiente salto.
- Dos paquetes de una conversación pueden ir por caminos distintos.

Routers: un salto a la vez



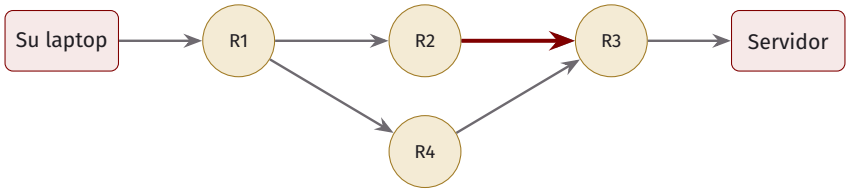
- El router mira el destino, consulta su tabla y entrega al **siguiente salto**.
- Cada router solo decide el siguiente salto.
- Dos paquetes de una conversación pueden ir por caminos distintos.

Routers: un salto a la vez



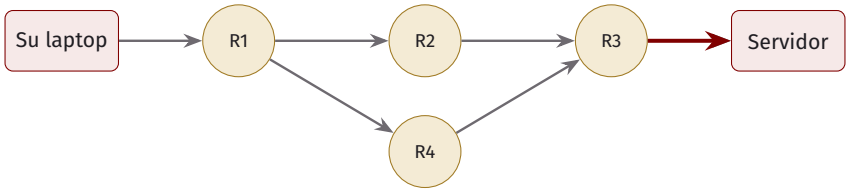
- El router mira el destino, consulta su tabla y entrega al **siguiente salto**.
- Cada router solo decide el siguiente salto.
- Dos paquetes de una conversación pueden ir por caminos distintos.

Routers: un salto a la vez



- El router mira el destino, consulta su tabla y entrega al **siguiente salto**.
- Cada router solo decide el siguiente salto.
- Dos paquetes de una conversación pueden ir por caminos distintos.

Routers: un salto a la vez



- El router mira el destino, consulta su tabla y entrega al **siguiente salto**.
- Cada router solo decide el siguiente salto.
- Dos paquetes de una conversación pueden ir por caminos distintos.

Intentar ver los saltos

2 de noviembre de 2026, depende de la red

```
$ traceroute -n -w 1 -q 1 -m 5 example.com
traceroute to example.com (104.20.23.154), 5 hops max, 60 byte packets
 1  172.17.0.1  0.026 ms
 2  *
 3  *
 4  *
 5  *
```

- `172.17.0.1`: la salida del contenedor, dentro de Docker.
- `*`: no llegó respuesta a tiempo. **No** significa necesariamente que el paquete se perdió: muchos equipos no responden a estas pruebas, y Docker puede impedir ver las respuestas.
- En su red puede salir algo muy distinto. El laboratorio no lo necesita.
- `example.com` es un nombre reservado para ejemplos.

Cada protocolo resuelve su parte

Pregunta	Quién responde
¿A qué computadora va?	IP, con la dirección IP
¿A qué programa de esa computadora va?	TCP, con el **puerto**
¿Llegó todo y en orden?	TCP: confirma y reenvía
¿Qué significan esos bytes?	La aplicación, como HTTP

- IP hace su mejor esfuerzo: un paquete se puede perder o llegar en otro orden.
- TCP abre una **conexión** y entrega una secuencia de bytes completa y en orden.
- Puerto: de 0 a 65535. Por costumbre, 80 para HTTP y 443 para HTTPS. El nuestro: 8014.

SECCIÓN 02

DNS

02



DNS: de nombres a números

- Las personas recuerdan `example.com`. Los routers solo entienden direcciones IP.
- Una agenda gigante, repartida en muchos servidores. Su computadora le pregunta a un **resolvedor**.

Paso	El resolvedor le pregunta a	Y le responde
1	Un servidor raíz	“Para `com`, pregúntale a estos”
2	Un servidor de `com`	“Para `example.com`, pregúntale a estos”
3	El servidor de `example.com`	“La dirección es esta”

DNS no trae la página

Solo responde con una dirección. Y las respuestas se guardan un tiempo en una cache.

`dig` y `localhost`

2 de noviembre de 2026, depende de la red, recortado

```
$ dig example.com A
;; ANSWER SECTION:
example.com.        377 IN  A   104.20.23.154
example.com.        377 IN  A   172.66.147.243

;; Query time: 95 msec
;; SERVER: 192.168.65.7#53(192.168.65.7) (UDP)
```

- `A`: una dirección IPv4. `377`: segundos que puede seguir en la cache.
- `SERVER`: el resolvedor, aquí el de Docker. Va por UDP: sin conexión ni reenvíos.
- Sin internet, `dig example.com` no obtiene respuesta. `localhost` sigue funcionando porque está en un archivo de la máquina, `/etc/hosts`.

SECCIÓN 03

HTTP

03



Un servidor propio, en dos terminales

- **Cliente** (*(client)*): pide. **Servidor** (*(server)*): espera pedidos y los responde.
- Codespaces: dividan la terminal. Docker: otra terminal en la laptop y `env/run.sh` otra vez.

terminal 1: se queda esperando

```
$ python3 server.py
Servidor en http://127.0.0.1:8014/ (Ctrl+C para detener)
```

terminal 2

```
$ curl -s http://127.0.0.1:8014/saludo
Buenos días, Perú
```



`curl -v`: la conversación

recortado; fecha y versión de curl cambian

```
$ curl -v http://127.0.0.1:8014/saludo
* Connected to 127.0.0.1 (127.0.0.1) port 8014
> GET /saludo HTTP/1.1
> Host: 127.0.0.1:8014
> User-Agent: curl/8.14.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: ServidorDelCurso/1.0
< Date: Mon, 02 Nov 2026 14:50:34 GMT
< Content-Type: text/plain; charset=utf-8
< Content-Length: 20
<
Buenos días, Perú
```

- `*` comentario de `curl`
- `>` lo que **mandó**
- `<` lo que **recibió**

URL

Dice el protocolo, la computadora, el puerto y la ruta. Como la computadora es un número, no hizo falta DNS.

Las partes

Petición

Línea: método (`GET` pide, `POST` manda datos), ruta, versión.

Encabezados (*(headers)*): `Host` es obligatorio en HTTP/1.1.

Una **línea vacía** marca el final.

Respuesta

Línea: versión, código, explicación.

Encabezados, y una línea vacía.

Cuerpo (*(body)*): el contenido.

`Content-Length: 20`

Son bytes: `Buenos días, Perú` y su salto de línea son 18 caracteres, pero `í` y `ú` ocupan 2 bytes.

Es texto de verdad

terminal

```
$ curl -si http://127.0.0.1:8014/saludo | od -c | head -2
0000000   H   T   T   P   /   1   .   1       2   0   0       O   K  \r
0000020  \n   S   e   r   v   e   r   :       S   e   r   v   i   d   o
```

- `-i` incluye los encabezados de la respuesta. `-s` oculta la barra de progreso.
- En HTTP/1.1 las líneas de encabezado terminan en **`\r\n`**, no solo en `\n`. Importa en el laboratorio.
- Con `/`, el cuerpo es HTML. Un navegador recibe ese texto y lo dibuja.
- Una página suele nombrar imágenes y estilos: más peticiones. Rara vez es una sola.

La respuesta, línea por línea

```
$ curl -si http://127.0.0.1:8014/saludo
```

```
HTTP/1.1 200 OK                                  \r\n
```

Línea de estado
http_status: código 200, clase 2xx

```
Server: ServidorDelCurso/1.0                     \r\n
Date: Mon, 02 Nov 2026 14:50:34 GMT              \r\n
Content-Type: text/plain; charset=utf-8          \r\n
Content-Length: 20                               \r\n
```

Encabezados
http_status cuenta 4

```
solo estos dos bytes                             \r\n
```

Línea vacía
aquí se detiene el primer ciclo

Cuerpo: 20 bytes, los que anuncia Content-Length

B	u	e	n	o	s	␣	d	í	a	s	,	␣	P	e	r	ú	\n
								2 bytes								2 bytes	

18 caracteres, contando el salto de línea. Esta última línea termina en \n, sin \r.

Códigos de estado

Clase	Significa	Ejemplos
1xx	Información, sigan	100 Continue
2xx	Funcionó	**200 OK**
3xx	Vaya a otro lugar	301 Moved Permanently
4xx	El **cliente** pidió algo mal	403 Forbidden, **404 Not Found**
5xx	Falló el **servidor**	**500 Internal Server Error**

Un 500 no quiere decir "ustedes hicieron algo mal"

Los 4xx apuntan al pedido. Los 5xx apuntan al servidor: reintentar quizá funcione más tarde.

El 404 y el 500 son respuestas normales

terminal

```
$ curl -si http://127.0.0.1:8014/nada
HTTP/1.1 404 Not Found
Content-Length: 22

No existe esa página
```

terminal

```
$ curl -si http://127.0.0.1:8014/error
HTTP/1.1 500 Internal Server Error
Content-Length: 32

Algo falló dentro del servidor
```

- Omitimos algunos encabezados.
- La red funcionó, y el servidor corrió y respondió.
- El código lo escribe un programa. `/error` da un 500 a propósito.
- Si la función que atiende se cae con un error, no se manda ningún 500.
- Con el servidor apagado **no hay código**: `curl -sS -i` da un error de conexión.

HTTP/2 y HTTP/3

Versión	Cómo viaja
HTTP/1.1	Texto sobre TCP
HTTP/2	Binario sobre TCP, con varias peticiones a la vez en una conexión
HTTP/3	Binario sobre QUIC, que corre sobre UDP y ya incluye el cifrado

2 de noviembre de 2026, depende de la red, recortado

```
$ curl -sI https://example.com/
HTTP/2 200
date: Mon, 02 Nov 2026 14:50:41 GMT
content-type: text/html; charset=utf-8
```

Conservan las mismas ideas: métodos, rutas, encabezados y códigos.

SECCIÓN 04

HTTPS

04

Un espía en el camino

spy.py con HTTP, recortado

```
--- cliente a servidor, 107 bytes
GET /saludo HTTP/1.1.
Host: 127.0.0.1:8015.
Cookie: sesion=abc123.
.

--- servidor a cliente, 147 bytes
HTTP/1.1 200 OK.
Content-Length: 20.
.

--- servidor a cliente, 20 bytes
Buenos d..as, Per..
```

- `spy.py` se pone en medio y escribe lo que pasa. Solo sirve en su computadora.
- **Cookie**: un dato que el sitio le pide guardar al navegador. Quien la copie puede hacerse pasar por ustedes.
- Un punto es un byte que no se puede imprimir. Los pedazos y sus cuentas pueden cambiar entre corridas.

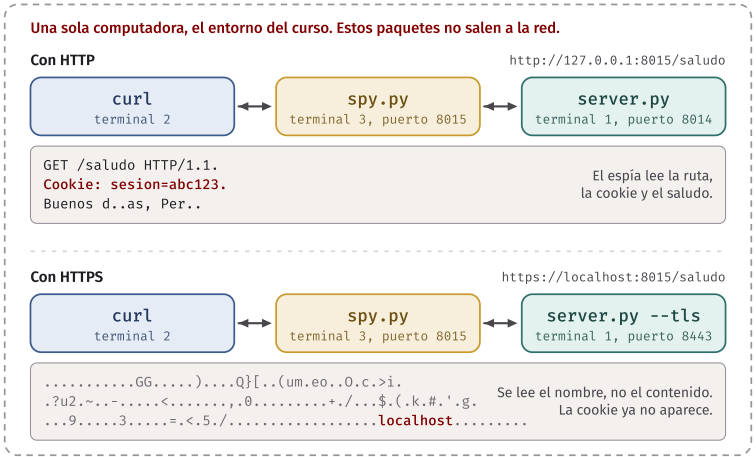
HTTPS = HTTP sobre TLS

spy.py con HTTPS: solo el comienzo, y cambia en cada corrida

```
--- cliente a servidor, 1569 bytes
...........GG.....)....Q}[..(um.eo..O.c.>i. ...a....:....xQE@Q
.?u2.~..-.....<.......,.0.........+./...$.(.k.#.'.g.
...9.....3.....=.<.5./..................localhost.........
.........................h2.http/1.1.........1.....6.4.........................
```

- **HTTP no cambia**. Antes de la petición, cliente y servidor acuerdan una clave secreta.
- Desde ahí todo viaja **cifrado**: no se puede leer sin esa clave, ni cambiar sin que se note.
- El cliente recibió lo mismo: `Buenos días, Perú`.
- Ya no se ven la ruta ni la cookie. **Se lee** `localhost`: el nombre va sin cifrar al comenzar.

El mismo espía, en una sola computadora



Qué ve quien administra la red

Ustedes abren `https://example.com/notas`.

Dato	¿Lo ve?	Por qué
Direcciones IP y puertos	**Sí**	Los routers los necesitan
El nombre `example.com`	**Normalmente sí**	Va en DNS y al comenzar TLS
Cuándo, y cuántos bytes	**Sí**	Se ve por el tráfico, sin leerlo

Qué ve quien administra la red

Ustedes abren `https://example.com/notas`.

Dato	¿Lo ve?	Por qué
Direcciones IP y puertos	**Sí**	Los routers los necesitan
El nombre `example.com`	**Normalmente sí**	Va en DNS y al comenzar TLS
Cuándo, y cuántos bytes	**Sí**	Se ve por el tráfico, sin leerlo
La ruta `/notas`	No	Va en la petición cifrada
Encabezados y cookies	No	También cifrados
Contraseñas y contenido	No	También cifrados

Hay técnicas nuevas que también cifran DNS y el nombre. No cuenten con ellas si no saben que están activas.

El certificado: con quién hablan

- Una conversación secreta con un impostor no sirve de nada.
- **Certificado**: “esta clave pertenece a este nombre”, firmado por una **autoridad de certificación** en la que el navegador ya confía.
- Si no hay ese respaldo, el cliente se niega a seguir:

recortado

```
$ curl -s -S https://localhost:8443/saludo
curl: (60) SSL certificate problem: self-signed certificate
```

Nuestro certificado de prueba, de `make cert`, no lo firmó ninguna autoridad. Con `--cacert build/cert.pem` le decimos a `curl` que confíe en él.

Lo que HTTPS no protege

- **Que el sitio sea honesto.** El candado o ícono del sitio solo dice que hablan cifrado con quien tiene ese nombre. Una página falsa con un nombre parecido también puede tener certificado.
- **Las puntas.** En el servidor, los datos se descifran. Si los guarda mal, HTTPS no ayuda. Tampoco ayuda con un programa malicioso en su computadora.
- **Una computadora administrada por otros.** Quien la administra puede hacerla confiar en un certificado suyo y leer su tráfico.

Idea clave

HTTPS protege el **contenido en el camino**, avisa si lo cambian y confirma el nombre. No oculta con quién hablan, ni dice si pueden confiar en ese alguien.

Abrir una página, paso a paso

pseudocódigo

```
algoritmo abrir_pagina(nombre, ruta)
    ip ← preguntar_a_dns(nombre)
    conexion ← abrir_tcp(ip, 443)
    canal ← iniciar_tls(conexion, nombre)
    respuesta ← pedir_http(canal, "GET", ruta)
    si codigo(respuesta) = 200 entonces
        dibujar(cuerpo(respuesta))
    sino
        escribir "Mostrar el error ", codigo(
     respuesta)
    fin si
fin algoritmo
```

Abrir una página, paso a paso

pseudocódigo

```
algoritmo abrir_pagina(nombre, ruta)
    ip ← preguntar_a_dns(nombre)
    conexion ← abrir_tcp(ip, 443)
    canal ← iniciar_tls(conexion, nombre)
    respuesta ← pedir_http(canal, "GET", ruta)
    si codigo(respuesta) = 200 entonces
        dibujar(cuerpo(respuesta))
    sino
        escribir "Mostrar el error ", codigo(
     respuesta)
    fin si
fin algoritmo
```

1. **DNS**: el nombre da la IP.

Abrir una página, paso a paso

pseudocódigo

```
algoritmo abrir_pagina(nombre, ruta)
    ip ← preguntar_a_dns(nombre)
    conexion ← abrir_tcp(ip, 443)
    canal ← iniciar_tls(conexion, nombre)
    respuesta ← pedir_http(canal, "GET", ruta)
    si codigo(respuesta) = 200 entonces
        dibujar(cuerpo(respuesta))
    sino
        escribir "Mostrar el error ", codigo(
     respuesta)
    fin si
fin algoritmo
```

1. **DNS**: el nombre da la IP.
2. **TCP**: abre la conexión con esa IP, puerto 443.

Abrir una página, paso a paso

pseudocódigo

```
algoritmo abrir_pagina(nombre, ruta)
    ip ← preguntar_a_dns(nombre)
    conexion ← abrir_tcp(ip, 443)
    canal ← iniciar_tls(conexion, nombre)
    respuesta ← pedir_http(canal, "GET", ruta)
    si codigo(respuesta) = 200 entonces
        dibujar(cuerpo(respuesta))
    sino
        escribir "Mostrar el error ", codigo(
     respuesta)
    fin si
fin algoritmo
```

1. **DNS**: el nombre da la IP.
2. **TCP**: abre la conexión con esa IP, puerto 443.
3. **TLS**: revisa el certificado y acuerda la clave.

Abrir una página, paso a paso

pseudocódigo

```
algoritmo abrir_pagina(nombre, ruta)
    ip ← preguntar_a_dns(nombre)
    conexion ← abrir_tcp(ip, 443)
    canal ← iniciar_tls(conexion, nombre)
    respuesta ← pedir_http(canal, "GET", ruta)
    si codigo(respuesta) = 200 entonces
        dibujar(cuerpo(respuesta))
    sino
        escribir "Mostrar el error ", codigo(
      respuesta)
    fin si
fin algoritmo
```

1. **DNS**: el nombre da la IP.
2. **TCP**: abre la conexión con esa IP, puerto 443.
3. **TLS**: revisa el certificado y acuerda la clave.
4. **HTTP**: pide la ruta, cifrado.

Abrir una página, paso a paso

pseudocódigo

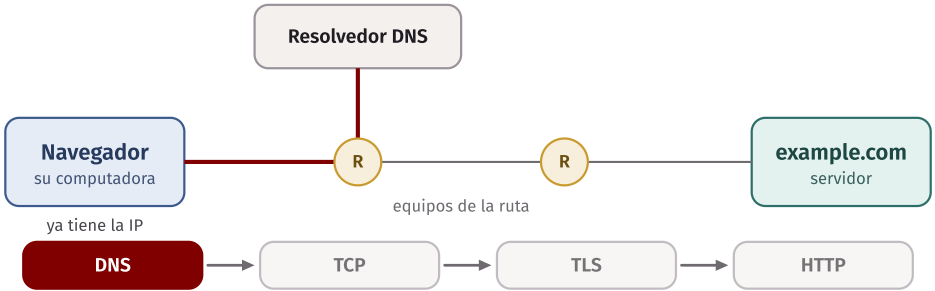
```
algoritmo abrir_pagina(nombre, ruta)
    ip ← preguntar_a_dns(nombre)
    conexion ← abrir_tcp(ip, 443)
    canal ← iniciar_tls(conexion, nombre)
    respuesta ← pedir_http(canal, "GET", ruta)
    si codigo(respuesta) = 200 entonces
        dibujar(cuerpo(respuesta))
    sino
        escribir "Mostrar el error ", codigo(
      respuesta)
    fin si
fin algoritmo
```

1. **DNS**: el nombre da la IP.
2. **TCP**: abre la conexión con esa IP, puerto 443.
3. **TLS**: revisa el certificado y acuerda la clave.
4. **HTTP**: pide la ruta, cifrado.

Con una visita reciente se saltan pasos. Con HTTP/3, QUIC reúne TCP y TLS sobre UDP.

Qué se ve en cada paso

Un sitio real en internet: https://example.com/. El laboratorio no necesita este camino.

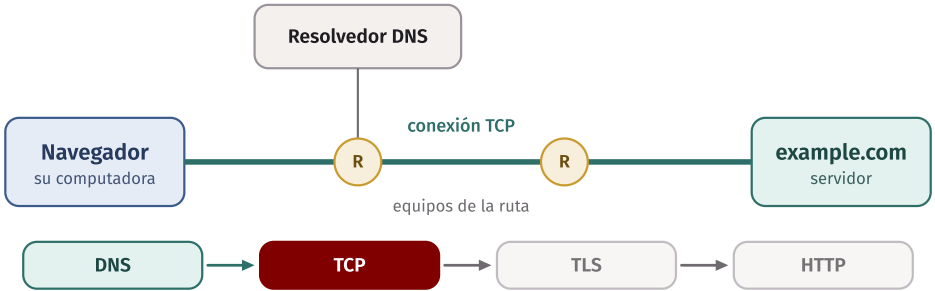


1. DNS: pregunta qué IP tiene el nombre. Necesita el nombre.

En el camino: normalmente, los equipos de la ruta pueden leer la pregunta.

Qué se ve en cada paso

Un sitio real en internet: https://example.com/. El laboratorio no necesita este camino.

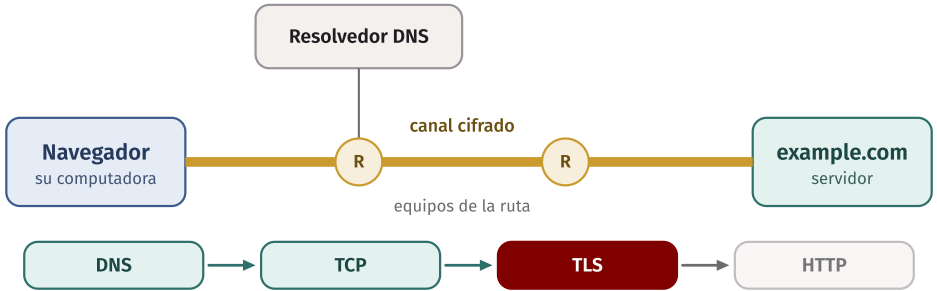


2. TCP: abre la conexión con esa IP, puerto 443. Necesita la IP.

En el camino: se ven la IP y el puerto.

Qué se ve en cada paso

Un sitio real en internet: https://example.com/. El laboratorio no necesita este camino.

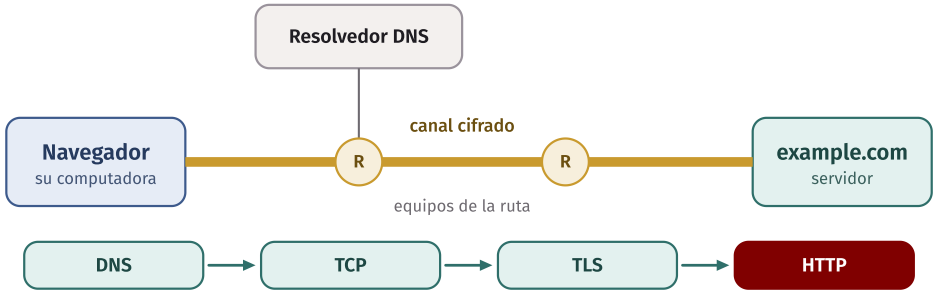


3. TLS: revisa el certificado y acuerda la clave. Necesita la conexión.

En el camino: se ve el nombre del sitio, que va sin cifrar al comenzar.

Qué se ve en cada paso

Un sitio real en internet: https://example.com/. El laboratorio no necesita este camino.



4. HTTP: pide la ruta y recibe la página. Necesita el canal cifrado.

Ruta, cookies y contenido: solo las dos puntas. A la vista: IP, puerto, cuándo y cuántos bytes.

SECCIÓN 05

Una mirada a Python

05



¿Por qué hay más de un lenguaje?

- `server.py` tiene unas 80 líneas de Python. En C pediría bastante más: red, cadenas con límites, memoria.
- A cambio, C suele correr más rápido y da control sobre cada byte.

Se necesita	Suele elegirse	Por qué
Sistema operativo, o equipo con poca memoria	C	Control y velocidad
Herramienta rápida, o procesar datos	Python	Poco código, muchas bibliotecas
Correr en el navegador	JavaScript	El navegador lo ejecuta

Compilado e interpretado

- **C con `gcc`**: traduce a código máquina. Corre la CPU, sin `gcc`.
- **Python con `python3`**: traduce a **bytecode**, y un ciclo escrito en C lo recorre instrucción por instrucción.
- Describen la herramienta, no el lenguaje: hay intérpretes de C.

show_bytecode.py

```
1  def add_one(x):
2      return x + 1
```

cambia con la versión de Python

```
$ python3 show_bytecode.py
  5           RESUME                   0

  6           LOAD_FAST_BORROW         0 (x)
              LOAD_SMALL_INT           1
              BINARY_OP                0 (+)
              RETURN_VALUE
```

El mismo programa: pseudocódigo

pseudocódigo

```
algoritmo es_primo(n)
    si n < 2 entonces
        devolver falso
    fin si
    d ← 2
    mientras d ≤ n div d hacer
        si n mod d = 0 entonces
            devolver falso
        fin si
        d ← d + 1
    fin mientras
    devolver verdadero
fin algoritmo
```

Es la función de la semana 7. `contar_primos` lee un límite y cuenta con un `para` de 2 hasta el límite.

En C y en Python

primes.c

```c
int is_prime(int n) {
    if (n < 2) {
        return 0;
    }
    int d = 2;
    while (d <= n / d) {
        if (n % d == 0) {
            return 0;
        }
        d = d + 1;
    }
    return 1;
}
```

primes.py

```python
def is_prime(n):
    if n < 2:
        return False
    d = 2
    while d <= n // d:
        if n % d == 0:
            return False
        d = d + 1
    return True
```

terminal

```
$ echo 100 | python3 primes.py
Hay 25 primos hasta 100
```

Qué cambió al leerlo

C	Python	Significa
`int n`	`n`	El tipo lo tiene el valor, y se revisa al correr
`{` y `}`	`:` y la sangría	La sangría **es** la estructura
`;`	Nada	El salto de línea termina la instrucción
`return 0;`	`return False`	Un tipo propio para verdadero y falso
`n <= limit; n++`	`for n in range(2, limit + 1):`	`range(a, b)` llega hasta `b - 1`
`scanf` con revisión	`int(input())`	Si no puede convertir, da un error
`printf("%d", count)`	`print(f"{count}")`	Reemplaza lo que hay entre llaves
`main`	Nada	Corre de arriba abajo
`for` con índice	`for grade in grades:`	Recorre los elementos, sin índice

La división no es igual

division.c

```c
printf("%d\n", 7 / 2);
printf("%.1f\n", 7.0 / 2);
```

division.py

```python
print(7 / 2)
print(7 // 2)
```

terminal

```
$ build/division
3
3.5
```

terminal

```
$ python3 division.py
3.5
3
```

Un error silencioso

En C, / depende del tipo. En Python, / da decimales y // es la división entera.

¿Cuál es más rápido? Midamos

una corrida: depende de la máquina

```
$ time build/primes_fast < limit.txt
Hay 78498 primos hasta 1000000

real    0m0.088s
user    0m0.068s
sys 0m0.015s
$ time python3 primes.py < limit.txt
Hay 78498 primos hasta 1000000

real    0m3.627s
user    0m3.512s
sys 0m0.014s
```

- `real`: reloj. `user` y `sys`: CPU en el programa y en el sistema.
- `primes_fast`: `-O2` y **sin** sanitizers, que hacen lento el programa.
- Python tardó unas 40 veces más. En otra corrida seguida, unas 25: no es una constante.
- Dos corridas no salen iguales.

Por qué un ciclo en Python suele ser más lento

`d = d + 1` con `-O2` en C: típicamente una instrucción. `gcc` ya sabe el tipo. En Python, en cada vuelta:

1. Leer la siguiente instrucción de bytecode, y decidir qué hacer.
2. Buscar qué tipo tiene el valor de `d`.
3. Elegir la suma de ese tipo, y revisar el resultado.
4. Guardar el resultado como un **objeto**, que lleva más que el número.

depende de la versión y de la máquina: `1` y `2 ** 100`, en bytes

```
$ python3 int_size.py
28
40
```

Un `int` de C ocupa 4 bytes en este entorno.

Pero no siempre gana C

50 000 números. Una corrida: depende de la máquina

```
$ time build/selection_sort_fast < sort_size.txt
menor: 0, mayor: 100001

real    0m2.192s
user    0m2.181s
sys 0m0.002s
$ time python3 sort_library.py < sort_size.txt
menor: 0, mayor: 100001

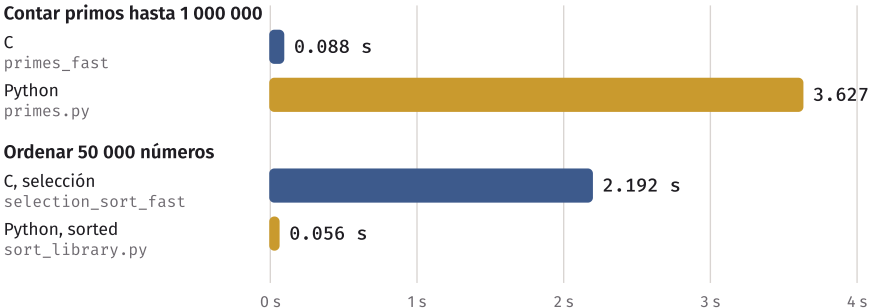
real    0m0.056s
user    0m0.020s
sys 0m0.003s
```

- Selección crece como N^2 (semana 13). `sorted` crece mucho más despacio.
- `sorted` está escrito en C.

Idea clave

El **algoritmo** importa más que el lenguaje. Antes de decir que algo es lento, mídanlo.

Las dos mediciones, a escala



Tiempo real de una corrida en la laptop donde se prepararon los apuntes. Depende de la máquina.

Un índice fuera de rango

C sin sanitizers: comportamiento indefinido, en esta corrida

```
$ echo 7 | build/index_out_plain
65535
```

C con sanitizers: recortado

```
$ echo 7 | build/index_out
errors/index_out.c:11:26: runtime error: index 7 out of bounds for type 'int [5]'
==93639==ERROR: AddressSanitizer: stack-buffer-overflow on address 0xffff7cc0005c at pc 0xaaaaaf5a14c8
```

En C, revisar el índice es responsabilidad de quien programa. Los sanitizers solo avisan si la ejecución pasa por ahí.

El mismo índice en Python

Python: ruta recortada

```
$ echo 7 | python3 errors/index_out.py
  File ".../errors/index_out.py", line 4, in <module>
    print(values[index])
          ~~~~~~^^^^^^^
IndexError: list index out of range
```

terminal

```
$ echo -1 | python3 errors/index_out.py
20
```

- Python revisa **cada** acceso a la lista: es parte de lo que cuesta.
- Dice el archivo, la línea y la causa, y se detiene antes de leer memoria ajena.
- Un índice negativo es válido en Python: cuenta desde el final.

Un entero que no cabe

C: comportamiento indefinido, en esta corrida

```
$ echo 13 | build/factorial
factorial.c:12:16: runtime error: signed integer overflow: 479001600 * 13 cannot be represented in type 'int'
13! = 1932053504
```

terminal

```
$ echo 13 | python3 factorial.py
13! = 6227020800
$ echo 25 | python3 factorial.py
25! = 15511210043330985984000000
```

- Los enteros de Python crecen todo lo que haga falta, mientras haya memoria.
- Por defecto no convierte a texto más de 4300 dígitos: un límite de impresión, no un overflow.

Un error de tipos, al correr

errors/late_error.py

```python
n = int(input())
if n > 0:
    print("positivo")
else:
    print(n + "x")
```

ruta recortada

```
$ echo 5 | python3 errors/late_error.py
positivo
$ echo -1 | python3 errors/late_error.py
Traceback (most recent call last):
  File ".../errors/late_error.py", line 6, in <module>
    print(n + "x")
          ~~^~~~
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Con la entrada 5 parece perfecto

C revisa los tipos al compilar. Python, solo si llega a esa línea. Las pruebas importan más cuanto menos revisa el compilador.

Lo que alguien resolvió por debajo

En C, ustedes	En Python
Llevan capacidad y `n` aparte	La lista conoce su tamaño, y crece con `append`
Pasan la longitud a cada función	`len(lista)` funciona en cualquier lugar
Terminan en `'\0'` y cuidan el buffer	Un texto sabe su largo
Comparan con `strcmp`	`==` compara el contenido
Revisan `scanf`, `fgets` y `fopen`	Si nadie lo maneja, un error detiene el programa
Revisan índices, o usan sanitizers	Cada acceso se revisa

Idea clave

Python es cómodo porque alguien resolvió por debajo lo que ustedes ya vieron en C: `python3` está escrito, en su mayor parte, en C. Se paga con velocidad, memoria y errores que aparecen al correr.

SECCIÓN 06

Laboratorio

06

Laboratorio: ver HTTP y resumirlo con C

1. **Ver HTTP**: `python3 server.py` en una terminal y `curl -v` en otra. Identifiquen las partes.
2. **El 404**: pidan `/nada` y `/error`. Después completen `lab/http_status.c`: `parse_status`, `is_blank_line` y dos ciclos con `fgets`.
3. **Comparar**: `time` con `primes_fast` y `primes.py`. Anoten el resultado **de su máquina**.

lo que debe dar

```
$ curl -si http://127.0.0.1:8014/saludo | build/lab/http_status
código: 200
clase: 2xx
encabezados: 4
cuerpo: 20 bytes
```

`make test-lab` levanta su propio servidor. De Python solo comprueba que corre.

Para la próxima clase

- En el laboratorio: **Práctica 4**. `lab/http_status.c` es trabajo autónomo.
- Leer los apuntes de la semana 15: inteligencia artificial y ética.

Preguntas

Ordenen lo que pasa al abrir `https://example.com/`: TLS, DNS, HTTP y TCP.
Con HTTPS, ¿qué puede ver quien administra la red del laboratorio, y qué no?
¿Qué error da C y qué error da Python con un índice fuera de rango?